A Suite of Software Tools for Managing a Large Parallel Programming Project

Paul A. Schwarz

Cornell Theory Center

May 1993

Abstract

A suite of software tools is presented for managing a large parallel programming project. The tools were selected recognizing that parallel program development is an iterative process and subject to mistakes and that software tools can be useful for maintaining source code flexibility and portability, tracking revisions, and analyzing variable usage and loop structure within a program. The tools discussed are: *make*, *cpp*, RCS, and FORGE 90. The concept of a *toy program* is introduced as a means for experimenting with a simpler version of an application program. Finally, the use of these tools and techniques is demonstrated as part of an optimization and parallelization effort for a scientific application program called *ZELIG*.

Introduction

The vast majority of all parallel programs in use today are modifications of existing serial programs. Few programs have been written from the outset for a parallel computer. As parallel computers become more readily available, parallel programming language constructs become standardized, and the collective parallel programming expertise of programmers expands, more parallel programs will be designed and coded. However, it is worth considering how one might improve the process of converting an application program from a conventional, uni-processor computer architecture to that of a scalable, multi-processor computer architecture such as the Kendall Square Research KSR1.

This report presents an overview of several programming tools and techniques that can be effectively combined and used to manage a large parallel programming project. We also present an example of a scientific application program for which we use these tools and techniques in order to help "parallelize" the program for execution on the Cornell Theory Center's (CTC) KSR1. With one exception, all of the tools discussed in the report are included in the standard UNIX programming environment or are freely available from network archives. They are general purpose tools and can be used for a variety of different applications. The objective of this report is not to discuss the details of how to use these tools and techniques—there are other sources that can better provide this information—but to show how these tools can be used together as part of integrated application development project. (For a list of sources, refer to the Conclusions and Bibliography.)

This report has three sections. The first discusses some programming and development considerations characteristic of a large parallel programming project. These considerations provide a context for evaluating and selecting program development tools and techniques. The second section presents five different tools and techniques and suggests how they can be used in developing a parallel program. The third section examines how these tools and techniques have been used in a project to develop a parallel implementation of the *ZELIG* forest simulation model for use on the CTC's KSR1 parallel computer (Urban 1990).

Programming Considerations

The tools and techniques discussed below have proven useful in the development of parallel programs. Each was selected to address one or more of the programming considerations discussed below. Although some them were originally designed for use with C programs and systems programming, they are also useful for developing Fortran applications.

Most scientific applications are continually modified and enhanced and passed from scientist to scientist and from graduate student to graduate student. The development issues faced by the professional programmer/analyst are largely the same issues that a scientist or student might face when working with a new, unfamiliar

program. At the CTC, we work with many different people involved with many different applications and are often confronted with unfamiliar programs. Given the range of such applications, we are constantly searching for new tools and techniques that can be used to analyze the structure of an unfamiliar program and to gain insight into how to restructure the program to maximize its efficiency on a multi-processor architecture. The tools and techniques discussed below are ones that have been found effective.

There are several programming and development issues that are characterize of the parallel program development process. Recognizing and appreciating the importance of these issues is prerequisite to evaluating and selecting tools to make the process more efficient.

1. Parallel program development is an iterative process

Developing a parallel program is an iterative process. Usually the primary motivation for developing a parallel program is to maximize the run-time performance so that larger problems can be solved. It is unlikely that a large, complex application can be fully parallelized and optimized in just one step. It is more likely that a strategy will be followed which includes a sequence of steps toward a production-ready, final program. For example, execution profiling may be useful in identifying which subroutines and loops consume the most execution time. A logical strategy for optimizing and parallelizing the program is to focus first on the most time-consuming subroutine or loop, then the next one, and so on. This figurative "do loop" is an integral component in parallel program development (Figure 1). Being able to practice this iterative approach efficiently is an important consideration when evaluating tools.

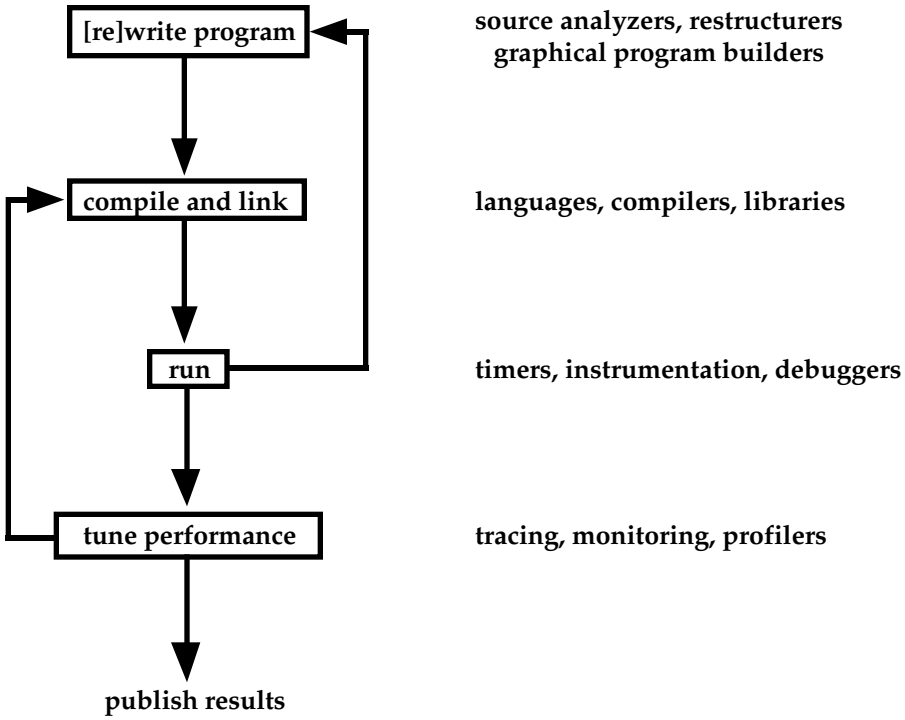
2. Source code flexibility and portability

Another programming consideration is the importance of maintaining source code flexibility and portability. It is very desirable to have a single version of the source code that can be compiled and run on many different machines. Maintaining one version of the program that runs on *workstation X*, and another version of the program that runs only on *parallel computer Y*, and third version that runs on *computer Z* is confusing and creates difficulties tracking bug fixes, enhancements, modifications, etc.

3. Revision history and documentation

Because developing a parallel program is a trial and error process in addition to being an iterative process, it is important to keep track of changes in the source code and to be able to backtrack if necessary. Fully optimizing and parallelizing a code can be a messy business. Often a code will require extensive restructuring in order to fully exploit an architecture and mistakes are inevitable.

Figure 1. Stages of Parallel Programming (from Bergmark (1993)).



(a) The Stages of Programming **(b) Tools for Parallel Programming**

4. Variable usage and loop structure

In order to parallelize an application program successfully, one must understand how a program references variables (especially arrays) and how its loops are structured. This understanding is crucial to determining which variables are local and which are global and to determining which loops have dependences. One method of developing this understanding is to print out the program and then mark up the printout with colored hi-lighters. However, software tools such as FORGE 90 are now available and can be used to address this fundamental parallel programming concern.

5. Start simply

Scientific application programs tend to be lengthy and complicated. Often they have many subroutines and functions and usually the essential data structures and algorithms are buried inside the program. We argue that the complexity and level of detail necessary in a production-ready program just obscure and complicate unnecessarily the job of parallelizing a program. Working with a simpler program makes it easier to experiment with different parallelization techniques and to evaluate their performance benefits. Devising a means to work first with a simple case before moving to a more complicated case can be an effective program development technique.

The Tools and Techniques

Each of the following five programming tools and techniques has been selected to address one or more of the programming and development considerations discussed above. The tools selected are: *make*, *cpp*, RCS, FORGE 90, and the use of toy programs.

1. make

The *make* facility is a compilation management tool and is an integral part of the UNIX programming environment. It can save time by eliminating unnecessary compiles and can ensure that the executable version of the program contains the most recent source code modifications. *make* can also guarantee that each subroutine and function is compiled and linked with the correct options.

make works best when each subroutine and function is stored in its own source file. The benefits of doing this are several. First, individual source files reduce the tedium associated with editing a single, large source file and locating a particular program unit. Second, individual source files eliminate the need to compile the whole program when only one routine is modified. Third, individual source files make it possible to use different compiler options e.g., debugging flags, on different program units. If a program has not already been separated into individual files, this can easily be done using the UNIX utility program *fsplit*.

In its simplest form, *make* uses a file called *makefile* that contains a list of *targets* (usually program units), describes the relationships between the program units called *dependencies*, and includes instructions for compiling and linking the targets. With a *makefile*, the task of compiling a program can be reduced from `f77 -O -o prog main.f sub1.f sub2.f -l`*lib* `-l`*anotherlib* to just `make`. If subsequently a source code change is made to `sub2.f`, only `sub2.f` is re-compiled; the rest of the program is just re-linked. During the iterative process of developing a parallel program, using *make* can be a real time and work saver. An example *makefile* is shown in Figure 2.

Using *make* to manage program compilation can also make it easier to use several of the other tools described below. Indeed, several of the tools are designed to be integrated with the *make* facility.

Figure 2. An example *makefile*.

```
# Example Makefile

# FC is the name of a "make" macro that contains the name
#   of the Fortran compiler ...
FC = f77

# FFLAGS contains a list of compiler options ...
FFLAGS = -O1 -kap

# PARA contains a list of loader options for parallelism on
#   the KSR1 ...
PARA = -para -lpresto
```

```
# OBJECTS contains the list of dependencies that comprise the
#   target "shalow" ...
OBJECTS = calc1.o calc2.o calc3.o calc3z.o cputim.o \
    inital.o shalow.o time0.o

# The following is a make "rule" which lists a "target", its
#   "dependencies" and the "command" for creating the target.
shalow: ${OBJECTS}
    ${FC} -o $@ ${OBJECTS} ${PARA}  -lpmon
```

2. cpp

cpp is a preprocessor and has several uses. In C programs, it is traditionally used to define simple statement macros and to include other files—usually header files. Since Fortran already has a `PARAMETER` statement and an `INCLUDE` statement, the most useful way to use *cpp* with Fortran programs is as a conditional compilation tool which enables a programmer to compile one group of Fortran statements within a single source file while not compiling another group. Which statements are actually compiled can be controlled at compile time.

The most important benefit of being able to compile a program conditionally is that it enables the programmer to, in effect, keep multiple versions of a program unit inside a single source file. For example, although the Fortran language description is a "standard," vendor implementations usually include several extensions and idiosyncrasies. It may make sense to take advantage of an extension in a particular situation on a particular architecture, but as a result, it is possible that the program will no longer run on a different computer. Instead of creating a separate source file containing the different version of the code, the programmer can use the preprocessor to direct the compiler to compile only those statements that the compiler supports.

The advantage of using a preprocessor for handling different language extensions also holds true for optimizations. Source code optimizations that may be appropriate for one computer architecture may be inappropriate for another architecture. Moreover, with an emphasis on program portability, multiple hardware platforms can be supported within a single source file. In other words, it is possible to have a single source file contain different optimizations for the KSR1, the RISC System/6000, and the ES/9000.

There are other preprocessors e.g., *m4*, but *cpp* is the most popular, and many Fortran compilers have built-in support for it. Sources files that have a `.F` suffix instead of the usual `.f` suffix are automatically passed through the *cpp* preprocessor before being processed by the compiler. *cpp* scans the file for preprocessor statements like `#ifdef` and `#endif`. `#ifdef` and `#endif` statements are the *cpp* equivalents of the Fortran `IF` and `ENDIF` statements. These statements are used to tell *cpp* which Fortran source statements to pass to the compiler. However, one additional piece of information is necessary to instruct *cpp* which source statements to include—the information necessary to evaluate the conditional. This information is usually contained in a simple boolean

symbol defined to be either 0 or 1. If the symbol is 1, then the Fortran source statements enclosed by the `#ifdef/#endif` pair will be compiled; otherwise the statements will not be compiled. The value of the symbol can be defined with either a `#define` preprocessor statement or with the `-Dsymbol` compiler flag. An example of using the *cpp* preprocessor is shown in Figure 3.

Figure 3. An example *cpp* preprocessor statement.

```
c     Diagnostics:
  624   continue
#ifdef DIAG_IO
        if (kr.eq.1.and.kc.eq.1.and.MOD(kyr,ilog).eq.0)
     2    write(LOGFILE,1103) ki, msp(ks), d, iht, mbc, algf,
     3    smgf, sfgf, ddf(ks), gf, dinc, nogro(kr,kc,ki)
 1103   format(10x,i3,1x,a4,f7.2,2i3,1x,4f5.2,f6.2,f7.2,i3)
#endif
   62   continue
      return
      end
```

RCS

The Revision Control System (RCS) is a set of utilities collectively used as a source code management tool for tracking software revisions. The two primary functions of RCS are maintaining source code revision logs and constructing different versions of a program. Although RCS was designed for multi-person programming teams for large development projects, it is, nevertheless, still useful for smaller projects. RCS is often not included in the standard UNIX distribution but is readily available from the Free Software Foundation (see Appendix A for more information). A similar package called the Source Code Control System (SCCS) is not free but is often included with UNIX distributions (especially System V-based distributions like AIX). In addition to being freely available for a variety of different machines, RCS has a simpler user interface than does SCCS.

In some respects, RCS is a source code repository which allows the programmer to *check in* and *check out* a file. To enter a file into an RCS repository , a file is *checked in* , and it is *checked out* in order to edit it or use it. RCS uses the differences between subsequent *check ins* plus programmer-supplied comments to track revisions and changes to the file. The revision log keeps track of the answers to the *who*, *what*, *when*, *where*, and *why* questions i.e., "who made the change?," "what was changed?," "when was the change made?," and "why was the change made?" An example RCS log file is shown in Figure 4.

The essential structure of the RCS repository is the revision tree. The revision tree is based on a tree structure, and each file has its own revision tree. The initial version of a file when it is first checked in is 1.1 and is called the *root*. Subsequent revisions are called

trunks, and a new development pathway is called a *branch* e.g., a new parallelization strategy.

Figure 4. An example RCS log file.

```
RCS file: RCS/grow.F,v
Working file: grow.F
head: 2.2
branch:
locks: strict
access list:
symbolic names:
comment leader: "c "
keyword substitution: kv
total revisions: 3;  selected revisions: 3
description:
This is version 2 of the GROW subroutine.  It calls the following functions:
HEIGHT, ALF, ALEAF, and DINCO.
----------------------------
revision 2.2
date: 1993/03/29 16:23:28;  author: schwarz;  state: Exp;  lines: +14 -14
Moved the third index of all 3D arrays to the first position so
that the arrays will be accessed in stride 1 order.
----------------------------
revision 2.1
date: 1993/02/23 16:08:47;  author: schwarz;  state: Exp;  lines: +6 -0
Added cpp #ifdef statements to conditionally compile diagnostic I/O
statements.
----------------------------
revision 2.0
date: 1993/01/29 15:05:01;  author: schwarz;  state: Exp;
This is the first version that I started working with.
=============================================================================
```

FORGE 90

FORGE 90 is the only commercial tool discussed in this report; it is very powerful. FORGE 90 is licensed and supported by the CTC for the RISC System/6000[1]. FORGE 90 is a sophisticated package and requires time to master, but the insight that can be gained from using the package—especially for large programs—can be invaluable. It is a very useful tool for analyzing program structure and variable usage—two critical elements needed for parallelizing a program. FORGE 90 also has the capability of actually inserting parallel constructs into a program—though this capability is currently limited to Cray Microtasking directives and subroutine calls that support message passing libraries such as PVM and Express.

[1] FORGE 90 timing libraries for execution profiling are also available for the ES/9000 and the KSR1 in addition to the RISC System/6000.

FORGE 90 is a stand-alone package that uses a sophisticated X window-based user interface. At the core of the package is a database that contains information derived from the application program. This information includes all variable references, control flow, and data flow. The database can be browsed and queried much like any other database, and variable usage can be traced across all the program units including variable aliasing via `COMMON` blocks, `EQUIVALENCE` statements, and `SUBROUTINE` argument passing. FORGE 90 also has a comprehensive program instrumentation and timing facility that expedites detailed routine and statement-level execution profiling.

A Fortran program is entered into a FORGE 90 database by the Fortran language parser. The parser scans all of the source files including `INCLUDE` files, does syntax checking, and then imports the information into the database. If the original source file is subsequently modified, FORGE 90 will automatically re-scan the file and update the database. Once the program is entered into a FORGE 90 database, it can then be cosmetically reformatted (code indentation, statement label re-sequencing, declaration statement reordering, etc.), instrumented, analyzed, and modified.

toy programs

The last tool recommended is not really a tool but a technique. Most production application programs are more complicated than just a collection of a few simple algorithms and data structures. Most programs include elaborate I/O schemes, error checking, initialization and boundary condition handling, check pointing, etc. The task of trying to parallelize such a program can be difficult and complex. Usually there are several ways to parallelize a program. The question then is: "how should a programmer select a parallelization strategy?" One solution is to use a *toy program*.

Although most application programs end up being quite complicated and are usually difficult to understand if you're not the original author, most programs do have just a few essential features. These features usually include one or more loops or subroutines, and one or more arrays. The basic technique is to extract these essential features from the larger, more complicated program and put them into a toy program. This toy program is much easier to understand and experiment with. A toy program can be used to investigate and evaluate different methods of parallelization and optimization. Because of its simplicity, the toy program can also be used as a means of establishing an asymptotic upper-limit in terms of performance.

When using a toy program, the shorter and simpler it is, the easier it is to work with—one or two pages of source code is ideal. The answer to the question of what to include in a toy program, though, is largely subjective. Tools like FORGE 90 and its profiling facility can be helpful in identifying the most time-consuming portions of the larger program. Often these loops or subroutines will be or contain the essential features of the entire program. All of the extra I/O, initialization, etc. may be extraneous.

Once the toy program is created, the experimenting can begin. Program execution times can be adjusted so that turn-around times are reasonable and compile times are short. Different parallelization constructs can be tried (e.g., KSR1 tile families, parallel

regions, pthreads, etc.), and *what if* games can be played. For example, "What if the tile size is changed?" or "What if a grab strategy is used instead of a slice strategy for tiling a particular loop?" With a toy program, those source code changes are easy to implement, and the results are easy to evaluate.

The experience gained from working with a simple program can then be directed back into the complete program along with some sort of expectation regarding run-time performance. Although it is unlikely that the same performance improvements achieved in the toy program will be achieved in the full program, it is likely that the relative performance ranking of the different optimization and parallelization techniques will be the same.

Integration of tools and techniques

Each of these tools addresses one or more of the parallel programming development issues discussed above. In addition, each of the tools can be used by itself or in combination with the other tools. They are not mutually exclusive. In fact, in many cases the maximum benefit from using the tools can only be achieved by using them in combination. *make*, *cpp*, and RCS have all been designed to work together. The *make* facility can streamline the process of repetitively editing and compiling a program. *cpp* can improve program flexibility and portability by keeping different versions in a single source file. RCS can track source code changes and simplify the task of backtracking if necessary. A single *makefile* can be used to *check out* a source file from RCS, define a preprocessor symbol (and hence control conditional compilation), compile the program unit, and then create an executable program. In the case of *xlf*, *fvs*, and *apf*—Fortran compilers for the IBM RISC System/6000 and ES/9000—*cpp* is not currently supported via the .F source file suffix. However, this capability can be easily emulated via a suffix rule in the *makefile* (see Appendix B). Many implementations of *make*, e.g., *GNU make* ,support RCS directly. For other implementations, another suffix rule can be written (Oram and Talbott 1991).

Although FORGE 90 is primarily used as stand-alone development tool, it, too, can be integrated with *make*, and *cpp*. FORGE 90 supports *cpp* symbols and statements and can use them for controlling program analysis. FORGE 90 can also generate a simple *makefile* for a program. And while FORGE 90 doesn't support RCS directly, it does provide support for SCCS. The *make* facility, *cpp*, and RCS, of course, can also be used with toy programs but their utility in such a situation should be evaluated on a case by case basis.

An Example

The optimization and subsequent parallelization of an application program is presented as an example of using the tools and techniques described above. The program is called *ZELIG*. *ZELIG* is a simulation model of forest dynamics and was developed by Dean Urban at Colorado State University (Urban 1990 and 1993). *ZELIG* is referred to as a gap model and is a descendant of an earlier model called *JABOWA* (Botkin et al. 1972). The model simulates forest dynamics by modeling the

establishment, annual growth, and mortality of each tree within a small (0.01 ha) grid cell corresponding to the region of influence of a canopy-dominant tree. *ZELIG* is comprised of a main program and 27 additional subroutines and functions containing about 3500 lines of Fortran code.

Dr. Urban and several others have been developing a parallel version of *ZELIG* to run on the CTC's KSR1 parallel computer. The model operates on a spatial grid, and the forest within each grid cell is simulated for a fixed number of years. The principal objective of the project was to develop a version of the model capable of simulating spatial grids of arbitrarily large size representing simulated forested landscapes. When we started the project, the entire program was in one source file. The first task was to separate the subroutines and functions using *fsplit* and then create a *makefile*. The program was originally developed on a Sun workstation, and a *makefile* made the process of porting the serial version of the program first to the RISC System/6000 and then to the KSR1 easier. Compilation on the KSR1 is relatively slow, but the machine has a parallel version of *make* which, when invoked with the `-jnumthreads` option, can compile several files simultaneously. This feature, alone, can greatly speed up global recompiles. The model used several Sun-specific Fortran I/O statements, and instead of deleting the statements or commenting them out, a *cpp* `#ifdef` statement was inserted to compile the statements only if the target computer were a Sun workstation. The *makefile* included a `CPPFLAGS` macro that contained a list of symbols each preceded by a `-D` to control the conditional compilation. The *makefile* also included commands to print out source files and emulate the *cpp* preprocessor when used on the RISC System/6000. The complete makefile is listed in Appendix B.

The next step after getting the original serial program to run was to analyze the program and then to suggest a strategy for developing a parallel version. For this step, we turned to FORGE 90. The program was imported into FORGE 90 and a FORGE database was created. Using FORGE, the entire program was instrumented to prepare the program for execution profiling. During a timing run, in addition to timing the entire program and each individual routine, FORGE collects data regarding how many times each statement is executed as well as how many times each routine is called and from where in the calling tree. These data are then used to calculate what percentage of the overall execution time is spent executing a particular statement, do loop, subroutine, etc. The results of the timing run were incorporated back into FORGE 90 and a dynamic call chain was created. With the dynamic call chain, FORGE 90 can create control and data flow diagrams of the program and can show COMMON block usage across each of the program units. Moreover, FORGE 90 can display which variables are used globally and which are used locally. In addition, not only can FORGE 90 identify which variables are global and which are local, but it can also show where the variables are set and where they are only referenced. FORGE 90's distributed memory parallelization facility was used to identify loop dependences and other parallelization inhibitors such as I/O.

FORGE 90 provided four important insights into the structure and operation of the *ZELIG* model. These insights were used to help optimize the program and were

precursors to actually parallelizing the model. The first insight stemmed from the model's loop structure and the choice of a parallelization strategy. Parallel programming theory suggests parallelizing a program at its highest functional level in order to maximize the application's inherent parallelism. This functional level is often called the problem domain, and the technique of structuring a parallel program at this level is referred to as domain decomposition (Carmona 1989). The problem domain is usually represented as a grid representing some physical structure which is then subdivided into subregions and can be assigned to individual processors. Therefore, the basic parallelization strategy we selected was to subdivide the forested landscape represented by the spatial grid and simulate each of the grid cells simultaneously. Within a single time step, the calculations within each grid cell were independent of those for other grid cells. At the end of the time step, data communication is required to simulate the flow of nutrients, seed propagules, disturbance, etc. among grid cells. The original model structure used a main program that contained a single loop which called the process routines (weather, tree mortality, growth, regeneration, and bookkeeping). Each of the process routines contained doubly-nested loops which performed the computations within each grid cell. To minimize the number of synchronization points and increase the program's granularity, the program was restructured so that the doubly-nested grid loops in each of the process routines were removed and replaced by one pair of doubly-nested loops in the main program.

Figure 5. The main programs from the original (left side) and restructured (right side) versions of the ZELIG model.

```
C                                                  C
    PROGRAM ZELIG                                      PROGRAM ZELIG
        .                                              include 'z2.inc'
        .                                          C   Main loop:
        .                                              call INITL
C   MAIN LOOP:                                         do 1 kyr=1,nyrs
    CALL INITL                                             call WEATHR
    DO 1 KYR=1,NYRS                                        do 2 kc=1,nrows
    CALL WEATHR                                            do 2 kr=1,ncols
    CALL MORTAL                                                call SOLWAT(kr,kc)
    CALL GROW                                                  call MORTAL(kr,kc)
    CALL REGEN                                                 call GROW(kr,kc)
    CALL BOOKS                                                 call REGEN(kr,kc)
    IF (MOD(KYR,IPRT).EQ.0) CALL PRINT                         call BOOKS(kr,kc)
    IF (MOD(KYR,IPCH).EQ.0) CALL PUNCH       2                 continue
  1    CONTINUE                                          call GRID
    STOP                                               1    continue
    END                                                stop
                                                       end
```

The second insight was the program's heavy use of a uniform random number generator for simulating biological processes such as tree regeneration and mortality and for environmental conditions like weather. Since the parallelization strategy was to

simulate each of the grid cells simultaneously, calls from each of the grid cells to the random number generator represented a critical section—a potential bottleneck. Fortunately, this problem had previously been encountered and solved using a parallel random number generator which was implemented on the KSR1 and based on the research of Percus and Kalos (1989). The calls to the parallel random number generator *prng* were enclosed inside a *cpp* `#ifdef/#else/#endif` combination so that *prng* was called when the program was compiled for the KSR1 and the serial random number generator was called otherwise.

The third insight from FORGE 90 was that the model performed I/O in each of the process subroutines. This prevented effective parallelization even though I/O on the KSR1 can be performed asynchronously. Fortunately, this problem too had an easy solution. It turns out that the I/O was mostly used for diagnostic purposes when debugging and was not necessary during production runs. We used another *cpp* `#ifdef` statement to compile the I/O statements only when the debugging information was needed.

The fourth insight gained from FORGE 90 concerned cache use. *ZELIG* made extensive use of three dimensional arrays to store data in which the first two array dimensions corresponded to the row and column coordinates within the spatial grid and the third dimension corresponded to a state variable inside the particular grid cell. An example is the array which stores the diameter of each tree growing in the grid cell. The problem with this array structure is that *ZELIG* did most of its computation by accessing the third dimension of these arrays. For each year in the simulation, the doubly-nested loop pair in the main program stepped through each grid cell and called five subroutines that each accessed the third dimension of several three dimensional arrays. Since Fortran stores arrays in column major order, this resulted in a memory stride of $n \times m$ where n and m were the dimensions of the grid. As the grid became larger, the stride became larger. This kind of memory access pattern was grossly inefficient and was a critical performance issue especially on the KSR1 where efficient cache use is crucial. To solve this problem, the three dimensional arrays were restructured so that the third dimension of each array became the first dimension. However, this global restructuring entailed a substantial amount of work in editing and modifying the program. In order to explore the performance consequences of restructuring the three dimensional arrays, we created a toy program.

The toy program was short—approximately one page long and is presented in Appendix C. It consisted of a main program that contained the same basic loop structure as *ZELIG* but without the weather and bookkeeping routines. Inside the doubly-nested grid loop pair, the toy program called a single subroutine whereas *ZELIG* called five subroutines. The subroutine contained a single do loop which looped on one of the dimensions of a three dimensional array. The investigation was done on the KSR1 and the toy program was timed for two different cases: one with the arrays structured as they were originally in *ZELIG* and the other case with the array and loops restructured. The timings showed that the latter case was roughly four times faster than

the former case and provided a quantitative justification for restructuring the three dimensional arrays and do loops in the complete program.

All of the program restructuring done to date has been serial optimization—preparatory work toward the ultimate goal of creating a parallel program. The optimizations reduced the CPU time used by the program. Future work includes integrating the structural changes made in the toy program back into the full program and analyzing the performance improvements in terms of reductions in CPU time. Performance improvements from parallelization will reduce the wall-clock time. At this time, the program restructuring work is still being done, but initial results suggest that CPU reductions of 25-50% are likely in the serial program.

Since the toy program is simpler and requires less time to run, it is now being used to investigate the consequences of using different KSR1 parallel constructs to execute concurrent iterations of the doubly-nested grid loop in the main program (Figure 5). These constructs include: *pthreads*, *parallel regions*, and *tile families* (KSR 1991). This investigation has resulted in several complications to the performance analysis—namely the overhead associated with each of the parallel constructs. These constructs represent a continuum of programmer control and complexity. Pthreads are low-level constructs and are the underlying mechanisms for the other constructs. Using them directly provides more control but demands more sophistication and understanding by the programmer. Consequently, the program overhead associated with using pthreads can be minimized. Parallel regions and tile families, on the other hand, are higher-level constructs and are easier for the programmer to implement but have higher overhead. The computational granularity of an application combined with the choice of construct can greatly influence a parallel program's performance. Unless the toy program is carefully tuned to match the computational grain size of the full program, it can be difficult to assess the speedups achieved by the different parallel constructs because of the overhead imposed by the constructs. This tuning requirement underscores the iterative and trial and error nature of parallel program development and suggests limits to the utility of toy programs. Program analysis, using FORGE 90, however, suggests that parallelizing the *ZELIG* model is possible but that the choice of parallel construct may be crucial. To date, the program restructuring, I/O removal, and the substitution of *prng* have removed dependences and inhibitors to parallelism. The next step is to examine the program overhead and select a KSR1 parallel construct that will best fit the granularity of the application.

Conclusions

The software tools and programming techniques discussed in this report have become integral components of the parallelization effort for the ZELIG model. This effort can be characterized as iterative and subject to mistakes and setbacks. Tools for tracking source code revisions and analyzing the program's loop structure and variable usage have been essential. These tools and techniques can also be applied to other parallelization efforts. Each of the tools discussed can help simplify the parallel program development process and can increase the efficiency of the programmer. Each

tool has a specific purpose and addresses one or more of the parallel programming and development issues. The *make* facility can reduce the tedium associated with iteratively developing a program, and *cpp* can help increase the flexibility and portability of a program. RCS can be used to track source code revisions and modifications. FORGE 90, the most sophisticated and powerful tool discussed, can offer critical insights into the structure and operation of a program. And finally, toy programs are a means for working with a simpler version of a program first before moving to a more complicated one. The use of toy programs as a development technique is promising but also poses challenges such as being able to tune them to match the characteristics of the complete program.

The availability of the software tools discussed above on the various CTC computing systems is shown in Table 1. More information on how to use the *make* facility can be found in two books published by O'Reilly & Associates. The first book *UNIX for FORTRAN Programmers* includes an excellent introductory chapter on *make*. The second book, which is also excellent, is devoted exclusively to the *make* facility and is called *Managing Projects with make*. UNIX *for FORTRAN Programmers* also demonstrates the use of the *cpp* preprocessor and includes a chapter on RCS. The latter book is highly recommended. A good introduction to FORGE 90 and its capabilities can be found in *FORGE 90 Baseline System User's Guide*.

Table 1. Software tool availability on Cornell Theory Center computing systems.

	make	**GNU make (gmake)**	**Compiler support for cpp**	**RCS**	**SCCS**	**FORGE 90**
RS/6000	yes	yes	no	no	yes	yes applic. & libs
ES/9000	yes	yes	no	no	yes	timing libraries only
KSR1	n.a.	yes	yes	yes	yes	timing libraries only
SP1	yes	yes	no	no	yes	no

Bibliography

Bergmark, Donna. 1993. Update on Tools for Parallel Programming at the CNSF. Theory Center Technical Report CTC93123. Cornell Theory Center, Ithaca, New York.

Botkin, D. B., J. F. Janak, and J. R. Wallis. 1972. Rationale, limitations, and assumptions of a northeastern forest growth simulator. IBM J. Res. Develop. **16**: 101-116.

Carmona, Edward A. 1989. Parallelizing a Large Scientific Code—Methods, Issues, and Concerns. In *Proceedings SUPERCOMPUTING '89*, November 13–17, 1989. The Association for Computing Machinery, New York.

Friedman, Richard. 1992. *FORGE 90 Baseline System User's Guide* Applied Parallel Research, Placerville, California.

Kendall Square Research. 1991. *KSR Parallel Programming*. Kendall Square Research Corporation, Waltham, Massachusetts.

Loukides, Mike. 1990. *UNIX for FORTRAN Programmers*. O' Reilly & Associates, Inc., Sebastopol, California.

Oram, Andrew and Steve Talbott. 1991. *Managing Project with make*. Second Edition. O' Reilly & Associates, Inc., Sebastopol, California.

Percus, Ora E. and Malvin H. Kalos. 1989. Random number generators for MIMD parallel processors. Journal of Parallel and Distributed Computing **6**: 477–497.

Urban, D. L. 1990. A versatile model simulate forest pattern: a user' s guide to ZELIG version 1.0. Environmental Sciences Department, Univ. Virginia, Charlottesville.

Urban, D. L. 1993. A User's Guide to ZELIG version 2—with notes on upgrades from version 1. Department of Forest Sciences, Colorado State University, Fort Collins.

Appendix A: Sources of publicly available software

GNU make and RCS are available from the Free Software Foundation.

Free Software Foundation, Inc.
675 Massachusetts Ave.
Cambridge, Massachusetts 02139
USA

The software is also available via anonymous ftp from *prep.ai.mit.edu* in the */pub/gnu* subdirectory. In addition, examples from the O' Reilly books can be *ftp*ed from: *uunet.uu.net*.

Appendix B: The *Makefile* for *ZELIG*

```
# Makefile for ZELIG ...

# To use, type "make xxx" where xxx is the name of the program to make.
# To see a list of valid targets type "make help".

# By convention, user-defined macros are lower case, and system-defined
# macros are upper case.

# Unless described explicitly, make uses default "suffix" rules for
# building targets.  As it turns out, these default rules usually do the
# right thing and are what you want.  So all you need to do is to define
# the dependencies between files, and make will do the rest.
```

```
# Compiler flags (-C for array bounds-checking, -O to optimize):
FFLAGS = -C -O2

# The MAKEFLAGS macro can be used to automatically turn on the parallel
#   option of GNU make on the KSR1.
MAKEFLAGS = -j3

# The CPP and CPPFLAGS macros allows the programmer to do conditional
# compilation via #ifdef/#endif statements and -D flags which are passed
# to the C preprocessor (cpp).  Generic f77 already knows how to do this,
# but unfortunately IBM's xlf, fvs, and apf compilers do not.  So I've
# written a new suffix rule to emulate this capability.  If you want to
# use the cpp, be sure to use the .F suffix for your source code files.

# CPP directive symbols:  (Use -Dsymbol to turn the feature on)
#   SUN -- Sun f77-specific statements
#   DIAG_IO -- diagnostic I/O
#   KSR1 -- KSR-specific statements
#CPP = /lib/cpp
#CPPFLAGS = -P
#CPPFLAGS = -P -DSUN
CPPFLAGS = -DKSR1

# Source code printing ...
LPR = enscript
LPRFLAGS = -2rG

# The following lines should be un-commented when direct support for
#   cpp is unavailable.
# Define some new suffix rules...
#.SUFFIXES    : .F

#.F.f  :
#  $(CPP) $(CPPFLAGS) $< > $*.f

#.F.o  :
#  $(CPP) $(CPPFLAGS) $< > $*.f
#  $(FC) $(FFLAGS) -c $*.f
#  -rm -f $*.f

# Source and object file definitions for ZELIG and support programs ...
all   = dirt solar weather grow esi gap z2

z2_srcs = z2.F initl.F weathr.F solwat.F books.F grow.F regen.F mortal.F \
          print.F punch.F grid.F dla.F wood.F height.F aleaf.F \
          dinco.F alf.F drtf.F fertf.F degdf.F amort.F smort.F \
          ran1.F ran2.F ran3.F gauss1.F gauss2.F gamma.F tline.F

z2_objs = $(z2_srcs:.F=.o)

d_src = dirt.F

d_obj = $(d_src:.F=.o)
```

```
s_src = solar.F

s_obj = $(s_src:.F=.o)

w_srcs = weather.F tline.F ran1.F gauss1.F gamma.F

w_objs = $(w_srcs:.F=.o)

t_srcs = tree.F wood.F height.F aleaf.F dinco.F \
         alf.F drtf.F fertf.F degdf.F

t_objs = $(t_srcs:.F=.o)

e_srcs = esi.F height.F aleaf.F dinco.F alf.F drtf.F fertf.F degdf.F

e_objs = $(e_srcs:.F=.o)

g_srcs = gap.F wood.F height.F aleaf.F dinco.F alf.F drtf.F fertf.F degdf.F \
         amort.F smort.F ran2.F gauss.F gamma.F tline.F

g_objs = $(g_srcs:.F=.o)

# The following prints out a list of targets when "make help" is issued.
help:
    @ echo ""
    @ echo "Makefile for ZELIG
    @ echo ""
    @ echo "List of valid targets:"
    @ echo "   all -- builds the whole shebang"
    @ echo "   z2 -- builds the zelig model"
    @ echo "   dirt -- builds the dirt support program"
    @ echo "   weather -- builds the weather support program"
    @ echo "   grow -- builds the grow support program"
    @ echo "   esi -- builds the esi support program"
    @ echo "   gap -- builds the gap support program"
    @ echo "   clean -- deletes executables and .o files"
    @ echo "   print_z2 -- prints the source files for the zelig model"
    @ echo ""

# Print source code files:
print_z2:
    $(LPR) $(LPRFLAGS) $(z2_srcs)

# Make ZELIG and all support programs:
zelig:  $(all)

# Link object files ...
z2:     $(z2_objs)
    $(FC) $(LDFLAGS) $(LOADLIBES) -o $@ $(z2_objs)

dirt:   $(d_obj)
    $(FC) $(LDFLAGS) $(LOADLIBES) -o $@ $(d_obj)
```

```
solar:  $(s_obj)
    $(FC) $(LDFLAGS) $(LOADLIBES) -o $@ $(s_obj)

weather:  $(w_objs)
    $(FC) $(LDFLAGS) $(LOADLIBES) -o $@ $(w_objs)

grow:  $(t_objs)
    $(FC) $(LDFLAGS) $(LOADLIBES) -o $@ $(t_objs)

esi:   $(e_objs)
    $(FC) $(LDFLAGS) $(LOADLIBES) -o $@ $(e_objs)

gap:   $(g_objs)
    $(FC) $(LDFLAGS) $(LOADLIBES) -o $@ $(g_objs)

# To clean up stuff ...
clean:
    - rm -f *.o $(all) core a.out

cleanf:
    - rm -f *.f

# Rules for building object files from source files ...

# ZELIG subroutines:
z2.o:  z2.F  z2.inc
initl.o:  initl.F  z2.inc
weathr.o:  weathr.F  z2.inc
solwat.o:  solwat.F  z2.inc
books.o:  books.F  z2.inc
grow.o:  grow.F  z2.inc
regen.o:  regen.F  z2.inc
mortal.o:  mortal.F  z2.inc
print.o:  print.F  z2.inc
punch.o:  punch.F  z2.inc
grid.o:  grid.F  z2.inc

# Soil moisture characters from texture (DIRT):
dirt.o:  dirt.F

# Solar radiation program SOLAR:
solar.o:  solar.F

# Stand-alone WEATHER and soil-water routine:
weather.o:  weather.F  weather.inc

# Support program GROW:
tree.o:  tree.F

# Support program ESI:
esi.o:  esi.F

# Support program GAP:
gap.o:  gap.F  gap.inc
```

```
# Functions ...
dla.o:  dla.F  z2.inc
height.o:  height.F
aleaf.o:  aleaf.F
wood.o:  wood.F
dinco.o:  dinco.F
alf.o:  alf.F
drtf.o:  drtf.F
degdf.o:  degdf.F
fertf.o:  fertf.F
amort.o:  amort.F
smort.o:  smort.F
tline.o:  tline.F

# Random number generators (mostly clones):

ran1.o:  ran1.F
ran2.o:  ran2.F
ran3.o:  ran3.F
gauss1.o:  gauss1.F
gauss2.o:  gauss2.F
gamma.o:  gamma.F
```

Appendix C: The toy program for *ZELIG*

```
C -- Toy program for ZELIG
C -- This version accesses memory in a more efficient manner
      PROGRAM z2toy
     parameter ( nyrs = 100, nz = 10000, nrows = 10, ncols = 10 )
     real blotz ( nz, nrows, ncols )
     common /block/ kyr, blotz

C     Main loop:
      write(*,*) 'Initializing data ...'
      call init
     write(*,*) 'Entering main loop ...'
      do 1 kyr=1,nyrs
       if ( MOD( kyr, 10 ) .eq. 0 ) write(*,10) kyr
   10   format('   kyr=', i4)

        do 2 kc = 1, ncols
        do 2 kr = 1, nrows
           call sub(kr,kc)
    2   continue
    1 continue

      kz = 1
      do kc = 1, ncols, 9
       do kr = 1, nrows, 9
```

```
         write (*,20) kz, kr, kc, blotz(kz, kr, kc)
20       format( ' kz=', i2, ' kr=', i2, ' kc=', i2,
2              ' blotz(kz,kr,kc)=', f10.1 )
    end do
  end do
  write(*,*) 'Finished!'

   end

   subroutine init
  parameter ( nyrs = 100, nz = 10000, nrows = 10, ncols = 10 )
  real blotz ( nz, nrows, ncols )
  common /block/ kyr, blotz

  do k = 1, ncols
     do j = 1, nrows
       do i = 1, nz
         blotz (i, j, k) = FLOAT( j * k )
       end do
     end do
  end do
  return
  end

   subroutine sub (kr,kc)
  parameter ( nyrs = 100, nz = 10000, nrows = 10, ncols = 10 )
  real blotz ( nz, nrows, ncols )
  common /block/ kyr, blotz

  do i = 1, nz
     blotz (i, kr, kc) = blotz (i, kr, kc) + FLOAT( kyr )
  end do
  return
  end
```